

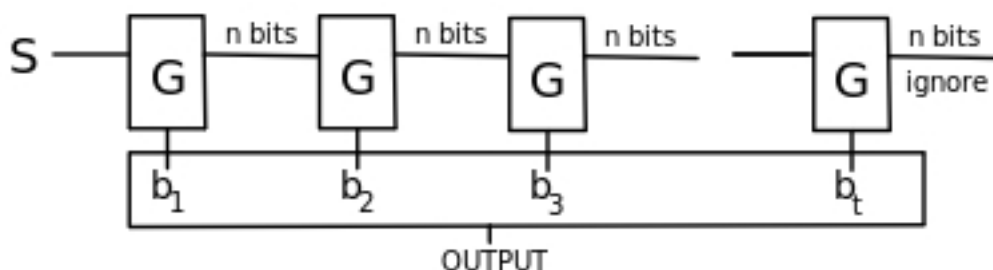
## 1 Expanding a PRG

In the last lecture we saw the definition of a Pseudorandom Generator (PRG) as a deterministic function that, given a seed of size  $n$ , outputs a pseudorandom string of length  $\ell(n)$ . We can ask the following question: how big can  $\ell(n)$  be? Is there any limit on how much pseudorandom data we can generate starting from a seed of a certain size? We will find out that if we can get a PRG with even *one bit* of expansion (i.e.,  $\ell(n) = n + 1$ ), then we can get a PRG with *any* polynomial output length.

**Theorem 1.1.** *Suppose that there exists a PRG  $G$  with output of length  $\ell(n) = n + 1$ . Then for any  $t(n) = \text{poly}(n)$  (where  $t(n) > n$ ), there exists a PRG  $G_t$  with output length  $t(n)$ .*

*Remark 1.2.* The size of the set  $\{G_t(s) : s \in \{0, 1\}^n\}$  is at most  $2^n$  (because  $G_t$  is deterministic), while the number of possible  $t(n)$ -bit string is  $|\{0, 1\}^{t(n)}| = 2^{t(n)}$ . The ratio of possible output strings that could actually be output by the PRG is at most  $2^n / 2^{t(n)} = 2^{n-t(n)}$ , which is absurdly small when  $t(n) \geq 2n$  (and even smaller when  $t(n) = n^{10}$ , say).

*Proof.* We will construct  $G_t(s)$  from  $G$ . Our construction will apply the function  $G(\cdot)$   $t(n)$  times, outputting one new bit at each step and reusing the other  $n$  bits of the previous step's output as a seed. See the following picture for intuition about the construction.



Formally,  $G_t$  is defined as follows (note that it always applies  $G$  on string of the same length,  $n$  bits):

---

### Algorithm 1 $G_t(s)$

---

```

if  $t = 0$  then
  return  $\varepsilon$  (the empty string)
else
  let  $(x|b) = G(s)$ , where  $x \in \{0, 1\}^n$ ,  $b \in \{0, 1\}$ 
  return  $b|G_{t-1}(x)$ 
end if
  
```

---

By construction,  $|G_t(s)| = t$ . We want to show that  $G_t$  is a PRG. The function clearly runs in polynomial time (each call to  $G$  can be resolved in polynomial time and we only use a polynomial number of steps), so what's left is to prove that  $\{G_t(U_n)\} \stackrel{c}{\approx} \{U_{t(n)}\}$ . We need to be careful: we already know that  $G$  is a PRG, but in our construction we are giving a *pseudorandom* seed to  $G$ , instead of a truly random seed. We will see that this fact will not affect the pseudorandomness of  $G_t$ . Intuitively, because no efficient algorithm can tell a pseudorandom seed apart from a random string, then in particular neither can  $G$ .

To prove that  $G_t$  is a PRG we will define a set of “hybrid experiments.” We will build a sequence of distributions, where the first is equal to our “real” construction  $G_t(U_n)$ , the last is equal to the “ideal” truly uniform distribution  $U_{t(n)}$ , and each consecutive pair of distributions are computationally indistinguishable. By the hybrid lemma, we conclude that  $\{G_t(U_n)\}$  and  $\{U_{t(n)}\}$  are computationally indistinguishable and that  $G_t$  is a PRG, thus proving the theorem.

To give some intuition about how we design the hybrid experiments, we imagine that instead of invoking the first  $G$  on  $n$  uniform bits, what if we replaced its output with  $n + 1$  truly uniform bits? Intuitively, these two cases should not be distinguishable, because  $G$  is a PRG. And then what if we replaced the first two invocations of  $G$ , and so on? Eventually, we would end up with  $t = t(n)$  truly uniform output bits, as desired.

Formally, the hybrid experiments are defined as follows:

- $H_0 = G_t(U_n)$
- $H_1 = U_1|G_{t-1}(U_n)$
- In general,  $H_i = U_i|G_{t-i}(U_n)$  for  $i \in \{0\} \cup [t]$
- $H_t = U_t$

We now show that for all  $i \in [t - 1]$ ,  $H_i \stackrel{c}{\approx} H_{i+1}$ . We will do this by using the simulation/composition lemma, and the fact that  $G$  is a PRG. For each  $i$ , we design a PPT “simulator” algorithm  $\mathcal{S}_i$  such that  $\mathcal{S}_i(G(U_n)) = H_{i-1}$ , and  $\mathcal{S}_i(U_{n+1}) = H_i$ . Since we know that  $G(U_n) \stackrel{c}{\approx} U_{n+1}$  from the fact that  $G$  is a PRG, the composition lemma implies that  $H_{i-1} \stackrel{c}{\approx} H_i$ .

We define  $\mathcal{S}_i$  as follows:

---

**Algorithm 2**  $\mathcal{S}_i(y \in \{0, 1\}^{n+1})$

---

parse  $y$  as  $(x|b)$  for  $x \in \{0, 1\}^n$ ,  $b \in \{0, 1\}$

**return**  $U_{i-1}|b|G_{t-i}(x)$

---

This algorithm clearly runs in polynomial time. We need to check that it maps  $G(U_n)$  to  $H_{i-1}$ , and  $U_{n+1}$  to  $H_i$ . First suppose that the input of  $\mathcal{S}_i$  comes from  $U_{n+1}$ :

$$\mathcal{S}_i(U_{n+1}) = U_{i-1}|b|G_{t-i}(x) = U_{i-1}|U_1|G_{t-i}(U_n) = U_i|G_{t-i}(U_n) = H_i.$$

Now suppose that the input comes from  $G(U_n)$ . By the definition of  $G_t$ , we can see the following:

$$\mathcal{S}_i(G(U_n)) = U_{i-1}|b|G_{t-i}(x) = U_{i-1}|G_{t-i+1}(U_n) = H_{i-1}.$$

This completes the proof. □

To recap, our proof of Theorem 1.1 took the following path:

- We defined a construction: the actual PRG  $G_t$  having a pseudorandom output of polynomial length.
- We defined the sequence of hybrid (“imaginary”) experiments. In each step, we replaced *one* “real” invocation of a crypto primitive ( $G(U_n)$ ) with its “ideal” counterpart ( $U_{n+1}$ ).
- We proved that consecutive pairs of hybrids are computationally indistinguishable, using the composition lemma and the security properties of the underlying primitives (i.e., that  $G$  is a PRG).
  - To apply the composition lemma, we defined a “simulator” (reduction) for each pair of adjacent hybrids and analyzed its behavior.

## 2 Obtaining a PRG

Thanks to Theorem 1.1, we know that all we need is to obtain a PRG with one extra bit of output. We describe a number-theoretic construction, due to Blum and Micali, of such an object.

### 2.1 Number Theory Background

**Theorem 2.1** (Euler’s theorem). *Let  $G$  be a finite abelian (i.e., commutative) multiplicative group. For every  $a \in G$ , we have  $a^{|G|} = 1$ .*

*Proof of Theorem 2.1.* Consider the set  $A = a \cdot G = \{ax : x \in G\}$ . Because  $G$  is a group,  $a$  is invertible, and we have  $A = G$ . Taking products over all elements in  $A = G$ , we have

$$\prod_{x \in G} (ax) = \prod_{x \in G} x.$$

Because  $G$  is commutative, the LHS is  $a^{|G|} \cdot \prod_{x \in G} x$ , and we can multiply by the inverse of the RHS to obtain  $a^{|G|} = 1$ .  $\square$

When  $G = \mathbb{Z}_p^*$  for a prime  $p$ , we have  $|\mathbb{Z}_p^*| = \varphi(p) = p - 1$ , so we obtain the following corollary:

**Corollary 2.2** (Fermat’s “little” theorem). *Let  $p$  be a prime. For any  $a \in \mathbb{Z}_p^*$ , we have  $a^{p-1} = 1 \pmod{p}$ .*

The following structural theorem will be very useful. (Its proof is elementary but rather tedious, so we won’t go through it today.)

**Theorem 2.3.** *Let  $p$  be a prime. The multiplicative group  $\mathbb{Z}_p^*$  is cyclic, i.e., there exists some generator  $g \in \mathbb{Z}_p^*$  such that  $\mathbb{Z}_p^* = \langle g \rangle := \{g^1, g^2, \dots, g^{p-1} = 1\}$ .*

### 2.2 Discrete Logarithm Problem and One-Way Function

Theorem 2.3 leads naturally to the so-called *discrete logarithm problem*, which is: given  $y \in \mathbb{Z}_p^*$  (and prime  $p$  and generator  $g$  of  $\mathbb{Z}_p^*$ ), find  $\log_g y$ , i.e., the  $x \in \{1, \dots, p-1\}$  for which  $y = g^x \pmod{p}$ . This problem is believed to be infeasible for large values of  $p$ .

**Conjecture 2.4** (Discrete logarithm assumption). Let  $S(1^n)$  be a PPT algorithm that outputs some prime  $p$  and generator  $g$  of  $\mathbb{Z}_p^*$ . For every non-uniform PPT algorithm  $\mathcal{A}$ ,

$$\Pr_{(p,g) \leftarrow S(1^n), y \leftarrow \mathbb{Z}_p^*} [\mathcal{A}(p, g, y) = \log_g y] = \text{negl}(n).$$

We would like to design a collection of OWFs based on the discrete logarithm assumption. The collection is made up of the functions  $f_{p,g} : \{1, \dots, p-1\} \rightarrow \mathbb{Z}_p^*$  (for prime  $p$  and generator  $g$  of  $\mathbb{Z}_p^*$ ), defined as

$$f_{p,g}(x) = g^x \pmod{p}.$$

Moreover, these functions are even *permutations* if we identify  $\{1, \dots, p-1\}$  with  $\mathbb{Z}_p^*$  in the natural way.

It is a tautology that the collection is one-way under the discrete logarithm assumption. It is also clear that we can efficiently sample from the domain of  $f_{p,g}$ . But we still need to check that  $f_{g,p}$  can be evaluated efficiently, and that  $(p, g)$  can be generated efficiently.

For the first, we use the standard “repeated squaring” technique for exponentiation, which requires  $O(|x|)$  multiplications modulo  $p$ . The solution to the second issue is not entirely straightforward. Given only the prime  $p$ , it is unknown (in general) how to find a generator  $g$  of  $\mathbb{Z}_p^*$  efficiently. However, given the *factorization* of  $p - 1$ , which can be generated along with  $p$ , it is possible: every element in  $\mathbb{Z}_p^*$  has order dividing  $p - 1$ , so  $g$  is a generator if and only if  $g^{(p-1)/q} \neq 1 \pmod{p}$  for every prime divisor  $q$  of  $p - 1$ . The number of non-generators is at most the sum of  $(p - 1)/q$  over all prime divisors  $q$  of  $p - 1$ , so the density of generators is typically large enough. An often-used special case is  $p = 2q + 1$  for prime  $q$ , for which there are  $q = (p - 1)/2$  generators. However, it is not even known whether there exist infinitely many such “Sophie Germain” primes of this form! (Empirically, though, they are abundant.)

## 2.3 Blum-Micali PRG

We now present a PRG that uses the ideas presented in the previous section. From Section 1 we know that if we have a PRG that is able to generate one extra bit of randomness, we can generate a polynomial number of pseudorandom bits. Our goal will be the following: we want to construct a PRG  $G_{p,g}: \mathbb{Z}_p^* \rightarrow \mathbb{Z}_p^* \times \{0, 1\}$ .<sup>1</sup>

Our solution is a function with the following form:

$$G_{p,g}(x) = (f_{p,g}(x) = g^x \pmod{p}, h(x)).$$

Note that  $f_{p,g}(x)$  performs the modular exponentiation function (which is a one-way under the discrete log assumption), while  $h: \mathbb{Z}_p^* \rightarrow \{0, 1\}$  is some function (yet to be defined) that provides the additional bit.

Looking at the function, we can make the following observation: if  $x \in \mathbb{Z}_p^*$  is chosen uniformly at random, then also  $f_{p,g}(x)$  is uniform (because  $f_{p,g}$  is a permutation). We still need to choose the function  $h$ , keeping in mind what we want from the function:  $h(x)$  should “look like a random bit,” *even given*  $f_{p,g}(x)$ . That is,  $h(x)$  should compute “something about  $x$ ” that  $f$  hides *completely*.

We can think of many possible candidates for  $h$ : apply the xor function to all the bit of  $x$ ; take the least significant bit of  $x$  (though you will show that this *does not* meet our requirements!); take the “most significant bit” of  $x$  (more precisely, test if  $x > \frac{p-1}{2}$ ). The last function will be the one we will use.

Let’s formalize the security properties we want from  $h(x)$ : given  $f(x)$ , no algorithm should be able to guess  $h(x)$  with much better than  $\frac{1}{2}$  probability.

**Definition 2.5** (Hardcore predicate). A predicate  $h: \{0, 1\}^* \rightarrow \{0, 1\}$  is *hard-core* for  $f$  if for all non-uniform PPT algorithms  $\mathcal{A}$ ,

$$\Pr_x[\mathcal{A}(f(x)) = h(x)] \leq \frac{1}{2} + \text{negl}(n).$$

*Exercise:* Prove that if  $h$  is hard-core for a one-way permutation  $f: D \rightarrow D$ , then

$$(f(x), h(x)) \stackrel{\mathcal{C}}{\approx} (U(D), U_1),$$

where  $x \leftarrow D$ . This means that  $G(x) = (f(x), h(x)) \in D \times \{0, 1\}$  is a PRG that expands by one bit.

Next time, we will show that under the discrete log assumption, the “most significant bit” predicate  $h(x) = [x > \frac{p-1}{2}]$  is hard-core for  $f_{p,g}$ .

---

<sup>1</sup>To be pedantic,  $G_{p,g}$  is a “collection” of PRGs, where the input seed comes from a set that depends on the function index  $(p, g)$ . It is easy to check that our construction from Section 1 is compatible with this collection  $G_{p,g}$ .